IBM Cognos Software Development Kit
Version 11.0.0

Dynamic Query Extensibility Developer Guide

Note

Before using this information and the product it supports, read the information in "Notices" on page 25.

Product Information

This document applies to IBM Cognos Software Development Kit Version 11.0.0 and may also apply to subsequent releases.

Contents

Introduction

This document is intended for use with IBM® Cognos® Analytics. It describes how to write functions that are called during the execution of dynamic queries in IBM Cognos Analytics - Reporting and in IBM Cognos Framework Manager.

Audience

To use the IBM Cognos Dynamic Query Extensibility Developer Guide effectively, you must be familiar with the following items:

- Dynamic query mode in IBM Cognos Analytics.
- Programming languages that can be used to write dynamic query extensibility functions..

Finding information

To find product documentation on the web, including all translated documentation, access IBM Knowledge Center (http://www.ibm.com/support/knowledgecenter).

Forward-looking statements

This documentation describes the current functionality of the product. References to items that are not currently available may be included. No implication of any future availability should be inferred. Any such references are not a commitment, promise, or legal obligation to deliver any material, code, or functionality. The development, release, and timing of features or functionality remain at the sole discretion of IBM.

Accessibility features

Consult the documentation for the tools that you use to develop applications to determine their accessibility level. These tools are not a part of this product.

IBM Cognos HTML documentation has accessibility features. PDF documents are supplemental and, as such, include no added accessibility features.

Chapter 1. Overview of dynamic query extensibility

Dynamic query extensibility provides a mechanism for extending the functionality of the dynamic query engine in IBM Cognos Analytics by adding functions that can be evaluated locally by the dynamic query engine.

Dynamic query extensibility is useful under the following conditions:

- Cognos built-in functions are not available to meet your business objectives.
- You do not want to, or cannot, put the functionality inside the database.

These functions can be written in SQL, in the Java™ programming language, or in JSR-223 scripting languages that can be run by the Java virtual machine. In this guide we provide examples of functions that use SQL and the Java programming language.

The dynamic query extensibility framework in Cognos Analytics is based upon the ISO/IEC 9075-13 SQL Routines and Types using Java Programming Language (SQL/JRT) extension to the SQL standard. This extension allows SQL applications to invoke static Java methods as routines.

Types of dynamic query extensibility functions

You can write three types of dynamic query extensibility functions.

Scalar functions
: Scalar functions accept zero or more arguments and return a single value, which may be a null value. Scalar functions are invoked in the same way as built-in dynamic query functions, such as `abs` or `floor`.. They are referenced in expressions in models and reports in the same manner as Cognos built-in or database scalar functions.

Aggregate functions
: Aggregate functions iterate over a set of values and return a single value, which may be a null value. Aggregate functions are invoked in the same way as built-in dynamic query aggregate functions, such as `avg` or `count`.. They can be included as part of an expression in an SQL statement or a report specification.

Table functions
: Table functions accept zero or more arguments and return a table of data. Table functions can be invoked in the `FROM` clause of an SQL statement and exposed in a query subject in a model or report.

Dynamic query extensibility supports overloading and polymorphism.

Implementing and using dynamic query extensibility functions

Implementing and using dynamic query extensibility functions in IBM Cognos Analytics is a collaborative effort between functions writers, system administrators, and report authors.

Function writer

The function writer performs the following tasks:

1. Writes and tests the code that implements the functions.
2. Creates deployment descriptor files that specify the interface between the functions and the dynamic query engine in the IBM Cognos Analytics server.
3. Package the class files and script files implementing your functions, and deployment descriptor files in a Java Archive (JAR) file.

These tasks are described in detail in Chapter 2, “Creating dynamic query extensibility functions,” on page 3.

System administrator

The system administrator deploys the Java Archive files to the Cognos Analytics server. This task is described in Chapter 3, “Deploying dynamic query extensibility functions,” on page 11.

Report author

The report author invokes the functions in expressions used in models and reports. The sample functions described in “Writing dynamic query extensibility functions” on page 3 include examples of how these functions are used in expressions.

Sample dynamic query extensibility functions

An installation of Cognos Analytics includes sample dynamic query extensibility functions.

A Java archive, `samples.jar`, is installed in `<`*`installation_location`*`>\v5dataserver\lib\ext`. The Java archive also includes the Java source files.

The sample functions are described in “Writing dynamic query extensibility functions” on page 3

Chapter 2. Creating dynamic query extensibility functions

The following topics illustrate the process for creating dynamic query extensibility functions. Examples are shown that use SQL and the Java programming language.

Writing dynamic query extensibility functions

There are three categories of dynamic query extensibility functions, scalar functions, aggregate functions, and table functions, that are described in the following topics.

Writing scalar functions

Scalar functions accept zero or more arguments and return a single value, which may be a null value. They are invoked in the same way as any scalar function in a dynamic query or a database query. They can be included as part of an expression in a model, a report specification. or a Cognos SQL statement.

Scalar functions can be written in SQL or the Java programming language. Java scalar functions are implemented as static methods of a class. The input parameters and returned value can be any Java type that maps to a supported dynamic query extensibility data type. See Appendix A, "Data type conversions from JDBC/SQL data types to Java data types," on page 17 for a list of supported data type mappings.

SQL scalar function example

SQL scalar functions are contained in a deployment descriptor file. This is an example of an SQL scalar function that convert temperatures from Celsius to Fahrenheit.

```
CREATE FUNCTION CELSIUS_TO_FAHRENHEIT(C INTEGER)
RETURNS FLOAT
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
RETURN (C * 9) / 5 + 32;
```

This function can be used in a SQL statement like any other scalar function. For example,

```
SELECT CELSIUS, CELSIUS_TO_FAHRENHEIT(CELSIUS) FAHRENHEIT
FROM TEMPERATURES
```

This query produces the following result.

Table 1: SQL scalar function result

CELSIUS	FAHRENHEIT
0	32.0
100	212.0

A screenshot using this sample function is shown here.

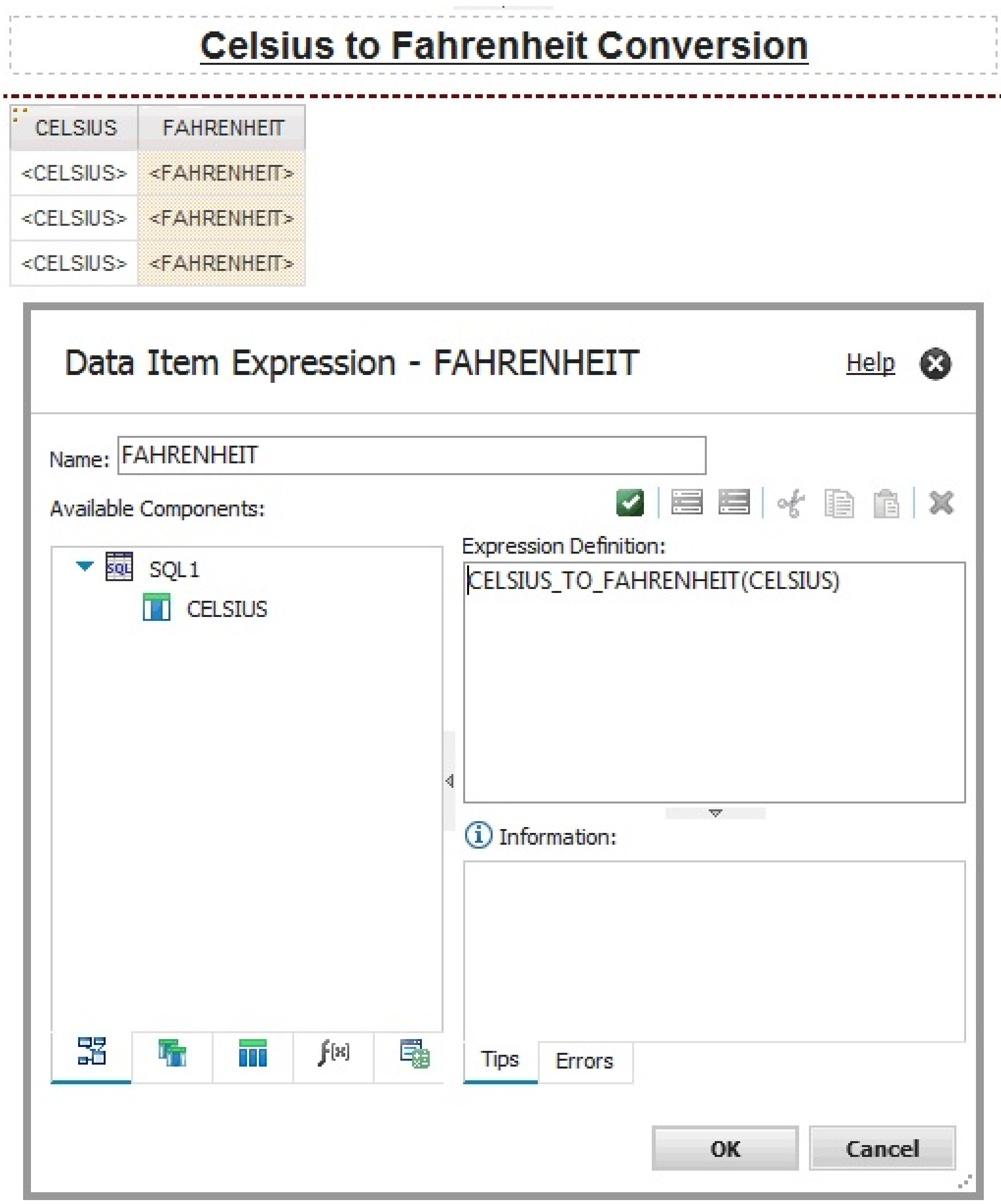

Figure 1: Sample function use in Cognos Reporting

Java scalar function example

This Java example formats currency based on country codes. The Java code is included in the `Format.java` sample program.

The following snippet in a deployment descriptor file is associated with this example.

```
CREATE FUNCTION formatCurrency(V DECIMAL(10,2), LANG VARCHAR(32), COUNTRY
VARCHAR(32))
RETURNS VARCHAR(32)
LANGUAGE JAVA
PARAMETER STYLE JAVA
EXTERNAL NAME 'thisjar:udf.samples.Format.formatCurrency';
```

The external name is the fully qualified method name (package.class.method) that contains the implementation of the logic for the function

This function can be used in a SQL statement like any other scalar function. For example,

```
SELECT PNAME, PRICE, formatCurrency(PRICE, 'en', 'US') FORMATTED_PRICE
FROM PRODUCTS
```

This query produces the following result.

Table 2: SQL scalar function result

PNAME	PRICE	FORMATTED_PRICE
Bolt	1.40	$1.40
Screw	1.50	$1.50

Writing aggregate functions

Aggregate functions iterate over a set of values, sharing the same values in a set of grouping columns in a GROUP BY or PARTITION clause, and return a single value per group or partition, which may be a null value. They are invoked in the same way as built-in dynamic query aggregate functions. They can be included as part of an expression in an SQL statement or a report specification.

Java aggregate functions

Java aggregate functions must contain at least the following methods, with the exception of the remove method, which is optional

initialize

The initialize method must return an instance of a class that implements the `java.io.Serializable` interface. The dynamic query engine uses this method to initialize the computation of the aggregation. This method is invoked once for each group or partition that the dynamic query engine is aggregating. A state object is returned.

The method signature is

```
public static Serializable initialize()
```

iterate

The iterate method accumulates the aggregate values and is invoked once for each value in the group that is being aggregated. The dynamic query engine calls this method after calling the initialize method. The implementation of this method should update the state of the instance to reflect the accumulation of the argument value being passed in.

The first parameter of the iterate method is the object returned by the initialize method.

The method signature is

```
public static void iterate(Serializable state, v <type> [, v <type>, ...])
```

remove

The remove method removes a value from the aggregation. The implementation of this method should update the state of the instance to reflect the accumulation of the argument value being passed in. For an example where this method is required, see Example where the remove method is required.

The first parameter of the remove method is the object returned by the initialize method.

The method signature is

```
public static void remove(Serializable state, v <type> [, v <type>, ...])
```

getResult

This method returns the current result of the aggregation.

The parameter of the getResult method is the object returned by the initialize method.

The method signature is

```
public static <type> getResult(Serializable state)
```

terminate

This method completes the aggregate computation and releases the resources used by the function.

The parameter of the terminate method is the object returned by the initialize method.

The method signature is

```
public static void terminate(Serializable state)
```

Java aggregate function example

This Java aggregate function performs progressive multiplication on all of its input values.. The Java code is included in the `Multiply.java` sample program.

The following lines in a deployment description file are associated with this example.

```
CREATE AGGREGATE multiply(n INTEGER)
RETURNS DOUBLE PRECISION
LANGUAGE JAVA
PARAMETER STYLE JAVA
EXTERNAL NAME 'thisjar:udf.samples.Multiply';
```

This aggregate can be used in a SQL statement like any other aggregate function or window function.

For example, consider the following sample data for the SEQUENCES table.

Table 3: SEQUENCES table

SEQUENCE	VALUE
S1	32
S1	8
S2	10
S2	20
S2	4

We can use this aggregate as a standard aggregate within a GROUP BY query as follows:

```
SELECT SEQUENCE, MULTIPLY(VALUE) RESULT
FROM SEQUENCES
GROUP BY SEQUENCE
```

This query produces the following result.

Table 4: Aggregate query result

SEQUENCE	RESULT
S1	256
S2	800

We can also use this aggregate as a window function. For example,

```
SELECT SEQUENCE, VALUE, MULTIPLY(VALUE) OVER () RESULT
FROM SEQUENCES
```

This query produces the following result.

Table 5: Window query result

SEQUENCE	VALUE	RESULT
S1	32	256
S1	8	256
S2	10	800
S2	20	800
S2	4	800

Example where the remove method is required

Consider the `MULTIPLY` aggregate defined previously. It is called in the following SQL snippet.

```
SELECT SEQUENCE,
  VALUE,
  MULTIPLY(VALUE) OVER (
    PARTITION BY SEQUENCE
    ORDER BY VALUE ROWS BETWEEN 1 PRECEDING AND CURRENT ROW
  ) RESULT
FROM SEQUENCES
```

This query produces the following result.

Table 6: Example using the remove method

SEQUENCE	VALUE	RESULT
S1	8	8.0
S2	32	256.0
S3	4	4.0
S4	18	40.0
S5	29	200.0

In order for this query to work, the optional remove method for the aggregate must be implemented. As the sliding window changes (`ROWS BETWEEN 1 PRECEDING AND CURRENT ROW`) for each row processed, this method is invoked to remove the value leaving the window. For example, assume the current window for within partition S2 consists of the values (4, 10). When the window moves to values (10, 20), the remove method is invoked to remove the value 4. The iterate method is then called to add the value 20 to the aggregate state.

Writing table functions

Table functions accept zero or more arguments and return a table of data. Table functions can be invoked in the FROM clause of a Cognos SQL statement.

SQL table function example

SQL table functions are contained in a deployment descriptor file. This is an example of an SQL table function that produces a result set consisting of parts and supply information

```
CREATE FUNCTION parts_supplied()
RETURNS TABLE(PNO CHAR(2), PNAME CHAR(10), SNO CHAR(2), QTY INTEGER)
LANGUAGE SQL
PARAMETER STYLE SQL
READS SQL DATA
DETERMINISTIC
RETURN
  SELECT P.PNO, P.PNAME, SP.SNO, SP.QTY
  FROM PARTS P, SUPPLY SP
  WHERE P.PNO = SP.PNO;
```

This table function can be used in the FROM clause of a SQL statement. For example,

```
SELECT PNO, PNAME, SNO, QTY
FROM TABLE( parts_supplied() ) T
WHERE QTY > 200
```

Java table functions

Java table functions are implemented as static methods whose return type is an object of a class that implements the `java.sql.ResultSet` interface. The input parameters can be any primitive types or objects that map to a supported dynamic query extensibility data type. See Appendix A, "Data type conversions from JDBC/SQL data types to Java data types," on page 17 for a list of supported data type mappings.

Java table function example

This function enumerates all of the locale information for the currently running Java Runtime Environment. The function returns a row for each locale consisting of the country, language, country code, language code, and currency code. The Java code is included in the `Locales.java` sample program.

The following lines in a deployment description file are associated with this example.

```
CREATE FUNCTION enumerateLocales()
RETURNS
TABLE(
COUNTRY VARCHAR(128),
"LANGUAGE" VARCHAR(128),
COUNTRY_CODE VARCHAR(32),
LANGUAGE_CODE VARCHAR(32),
CURRENCY_CODE VARCHAR(32)
)
LANGUAGE JAVA
PARAMETER STYLE JAVAEXTERNAL NAME
'thisjar:udf.samples.Locales.enumerateLocales';
```

This table function can be used in the FROM clause of a SQL statement like any other table function. For example

```
SELECT *
FROM TABLE( enumerateLocales() ) T
```

,

Creating the deployment descriptor file

The deployment descriptor file contains the source code for functions written in SQL, as well as input and output parameters and the method name for functions written in the Java programming language.

A simple deployment descriptor file is shown here, based on the SQL and Java functions described in "Writing scalar functions" on page 3.

```
SQLActions[] = {
  "BEGIN INSTALL
    CREATE FUNCTION CELSIUS_TO_FAHRENHEIT(C INTEGER)
    RETURNS FLOAT
    LANGUAGE SQL
    CONTAINS SQL
    DETERMINISTIC
    RETURN (C * 9) / 5 + 32;

    CREATE FUNCTION formatCurrency(V DECIMAL(10,2), LANG VARCHAR(32),
COUNTRY VARCHAR(32))
    RETURNS VARCHAR(32)
    LANGUAGE JAVA
    PARAMETER STYLE JAVA
    EXTERNAL NAME 'thisjar:udf.samples.Format.formatCurrency';
  END INSTALL"
}
```

Each deployment descriptor file can contain any number of entries for SQL and Java functions .

You should note the following issues when creating deployment descriptor files.

- The file name extension for the deployment descriptor file is ddl.
- Function names are case-insensitive.
- SQL keywords used as table or column names must be delimited as shown in the following example with the column name LANGUAGE.

```
CREATE FUNCTION enumerateLocales()
RETURNS
TABLE(
COUNTRY VARCHAR(128),
"LANGUAGE" VARCHAR(128),
COUNTRY_CODE VARCHAR(32),
LANGUAGE_CODE VARCHAR(32),
CURRENCY_CODE VARCHAR(32)
)
LANGUAGE JAVA
PARAMETER STYLE JAVAEXTERNAL NAME
'thisjar:udf.samples.Locales.enumerateLocales';
```

- When writing SQL functions, there can be any number of lines between the RETURN keyword and closing semi-colon(;).
- Square brackets, [and], must be replaced by the trigraph equivalent codes, which are ??(and ??), respectively.
- The input and output parameter types are Cognos SQL types. A table mapping Cognos SQL types to Java types can be found in Appendix A, "Data type conversions from JDBC/SQL data types to Java data types," on page 17
- A description of the structure of the deployment descriptor file in Backus–Naur Form can be found in Appendix C, "BNF grammar description for the deployment descriptor file," on page 21.

Creating the Java Archive file

The class files and .deployment descriptor files associated with dynamic query extensibility functions are packaged in a Java Archive (JAR) file.

The manifest for the JAR file contains a section for each data descriptor file in the package. The example manifest from the `samples.jar` file (see "Sample dynamic query extensibility functions" on page 2) is shown here.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.8.4
Created-By: jvmwi3260sr10-20111207_96808 (IBM Corporation)

Name: udf/samples/samples.ddl
SQLJDeploymentDescriptor: TRUE
```

Chapter 3. Deploying dynamic query extensibility functions

After a Java archive (.jar) packaging dynamic query extensibility functions is cretaed, a system administrator must deploy it to IBM Cognos Analytics server.

Dynamic query extensibility function .jar files are deployed in the `<installation_location>\v5dataserver\lib\ext` folder.

To add a new .jar file to this folder, copy the file to this location and then restart the Query Service in IBM Cognos Administration.

To update or remove an existing .jar file, stop the Query Service, remove or replace the existing .jar file, and then start the Query Service.

Chapter 4. Programming considerations when creating dynamic query extensibility functions

There are a number of prgramming issues you can consider when creating dynamic query extensibility functions. They are described in the following topics.

Implicit type conversion

Implicit type conversion, also known as coercion, is an automatic type conversion performed by the dynamic query engine. Implicit type conversion is used when argument types do not match the required parameter types of a function.

For example, consider the following function:

```
CREATE FUNCTION formatCurrency(V DOUBLE PRECISION) ...
```

Invoking this function with a value of 10, as in

```
SELECT formatCurrency(10)
FROM ( VALUES (0) ) T
```

will result in the SMALLINT value 10 being implicitly converted to a DOUBLE PRECISION value. The tables in Appendix B, “Implicit data type conversion rules,” on page 19 list all the allowable implicit type conversions

If an allowable implicit type conversion is not available, there are two possible outcomes.

- The dynamic query engines pushes the function into the native SQL generated for the target database. If the function is not valid or recognized, the user sees a database-specific error.
- Otherwise, a planning error occurs, as in:

```
XQE-PLN-0098 The vendor specific function "FOO" is not supported.
```

Overloaded functions

When creating functions, overloading is permitted. That is, functions with the same name can be defined that differ in the number and type of input parameters. This feature is also found in various programming languages.

For example, consider the following 3 functions, all named formatCurrency.

```
CREATE FUNCTION formatCurrency(V DOUBLE PRECISION)
...
CREATE FUNCTION formatCurrency(V DECIMAL(10,2))
...
CREATE FUNCTION formatCurrency(V FLOAT, LANG VARCHAR(32), COUNTRY
VARCHAR(32))
...
```

When formatCurrency is used in an expression, the dynamic query engine determines at run-time which version of the function to call, based on a best fit technique.

Function resolution for overloaded functions

When a function is invoked and there are multiple functions available with the same name, the dynamic query engine uses the following procedure to determine the function that fits best.

1. If a function with the same number and type of input parameters is available, that function is used.
2. Otherwise, a list of functions with the same number of input parameters that could be used with implicit type conversion is made.
3. If this list contains more than one function, the functions are ranked and the function with the lowest rank number is selected for execution.
4. If two or more functions share the lowest rank, the function that appears first in the data desciptor file is chosen.

Ranking functions

Each implicit type conversion is assigned a rank number. The rank number for an implicit conversion is equal to the difference in ordinal numbers for the original and converted data type. The ordinal numbers for each data type are shown in the following table.

Table 7: Implicit data type ordinal values

Data type	Ordinal
BOOLEAN	0
CLOB	1
DATE	2
TIME	3
TIMESTAMP	4
INTERVAL DAY-TIME	5
INTERVAL YEAR-MONTH	6
CHAR	7
VARCHAR	7
NCHAR	7
NVARCHAR	7
SMALLINT	8
INTEGER	9
BIGINT	10
FLOAT	11
DOUBLE PRECISION	12
DECIMAL	13

For example, the rank number for converting from a SMALLINT data type to a BIGINT data type is 10 - 8 = 2. Rank numbers are summed for each implicit type conversion to arrive at the overall rank number for the function.

Sample overloaded function resolution

Consider the following function definitions.

```
1. CREATE FUNCTION foo(V SMALLINT) ...
2. CREATE FUNCTION foo(V INTEGER) ...
3. CREATE FUNCTION foo(V1 SMALLINT, V2 INTEGER) ...
4. CREATE FUNCTION foo(V2 SMALLINT, V2 BIGINT) …
5. CREATE FUNCTION foo(V2 INTEGER, V2 INTEGER) ...
6. CREATE FUNCTION foo(V2 INTEGER, V2 BIGINT) ...
```

The function is invoked in an expression as `foo(C1,C2)` where `C1` is an INTEGER and `C2` is a SMALLINT. Functions 1 and 2 cannot be invoked due to a mismatch in the number of input parameters, and functions 3 and 4 cannot be used because an INTEGER cannot be cast to a SMALLINT. Function 5 has a rank number of 1 (SMALLINT to INTEGER cast) and function 6 has a rank number of 2 (SMALLINT to BIGINT cast). Thus function 6 will be used to evaluate the expression `foo(C1,C2)`.

Variadic functions

Variadic functions are functions that take a variable number of arguments. The dynamic query engine supports variadic functions.

Consider the following data descriptor file snippet that describes a scalar function that takes a variable number of integer arguments.

```
CREATE FUNCTION SUM_VALUES(IVAL INTEGER ...)
RETURNS BIGINT
LANGUAGE JAVA
PARAMETER STYLE JAVA
EXTERNAL NAME 'thisjar:udf.Arrays.sum';
```

The Java implementation of this function would look like this:

```
package udf;
public class Arrays {
  public static long sum(Integer... values) {
    long result = 0;
    for (Integer value : values) {
      result += value;
    }
    return result;
  }
}
```

This function could then be used as follows:

```
SELECT SUM_VALUES(10, 20) SUM, SUM_VALUES(C1, C2, C3, C4)
FROM T
```

Appendix A. Data type conversions from JDBC/SQL data types to Java data types

The following table shows how Cognos SQL data types are mapped to Java data types. Cognos SQL data types are the same as SQL data types and JDBC data types, except as noted in the table.

Table 8: Cognos SQL to Java data type mappings

Cognos SQL data type	**Java data type**
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NCHAR	String
NVARCHAR	String
LONGNVARCHAR	String
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
BOOLEAN	boolean, Boolean
SMALLINT	short, Short
INTEGER	int, Integer
BIGINT	long, Long
DECIMAL	java.Math.BigDecimal
NUMERIC	java.Math.BigDecimal
FLOAT	float, Float
REAL	float, Float
DOUBLE PRECISION (Note 1)	double, Double
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
CLOB	java.sql.Clob
BLOB	java.sql.Blob
INTERVAL DAY-TIME (Note 2)	String
INTERVAL YEAR-MONTH (Note 2)	String
ARRAY	java.sql.Array
STRUCT	java.sql.Struct
JAVA_OBJECT	Object
DATALINK	java.net.URL

Table 8: Cognos SQL to Java data type mappings (continued)

Cognos SQL data type	Java data type
XML (Note 3)	String
MULTISET	java.sql.ResultSet

Note 1
The JDBC data type is DOUBLE.

Note 2
There is no equivalent JDBC data type.

Note 3
The JDBC data type is SQLXML.

Appendix B. Implicit data type conversion rules

When resolving implicit type conversions, the dynamic query engine uses the following table to determine which conversions are allowed. If a data type is not shown in the first column of this table, no casting from it is possible.

Table 9: Implicit data type conversions

Original data type	Converted data type
SMALLINT	INTEGER, BIGINT, FLOAT, DOUBLE, DECIMAL
INTEGER	BIGINT, FLOAT, DOUBLE, DECIMAL
BIGINT	FLOAT, DOUBLE, DECIMAL
FLOAT	DOUBLE, DECIMAL
BOOLEAN	SMALLINT, INTEGER, BIGINT, FLOAT, DOUBLE, CHAR, VARCHAR, NCHAR, NVARCHAR
CHAR	VARCHAR, NCHAR, NVARCHAR
VARCHAR	CHAR, NCHAR, NVARCHAR
NCHAR	CHAR, VARCHAR, NVARCHAR
NVARCHAR	CHAR, VARCHAR, NCHAR
CLOB	CHAR, VARCHAR, NCHAR, NVARCHAR
TIME	TIMESTAMP, CHAR, VARCHAR, NCHAR, NVARCHAR
DATE	TIMESTAMP, CHAR, VARCHAR, NCHAR, NVARCHAR
TIMESTAMP	CHAR, VARCHAR, NCHAR, NVARCHAR
INTERVAL DAY-TIME	CHAR, VARCHAR, NCHAR, NVARCHAR
INTERVAL YEAR-MONTH	CHAR, VARCHAR, NCHAR, NVARCHAR

Appendix C. BNF grammar description for the deployment descriptor file

The structure of the deployment descriptor file can be described using the Backus–Naur Form as shown here.

```
deploymentDescriptor ::= <SQLACTIONS> <EQL> <LBRACE> <DQUOTE> actionGroup <DOUBLE_QUOTE>
<RBRACE>

actionGroup ::= installActions | removeActions

installActions ::= <BEGIN> <INSTALL> ddl <END> <INSTALL>

removeActions ::= <BEGIN> <REMOVE> <END> <REMOVE>

ddl ::= SQLInvokedRoutine ( SQLInvokedRoutine )*

SQLInvokedRoutine ::=
  ( <CREATE> SQLInvokedFunction | <CREATE> SQLInvokedAggregate | <CREATE> SQLInvokedProcedure )
  <SEMICOLON>

SQLInvokedFunction ::=
  <FUNCTION> SchemaQualifiedRoutineName SQLParameterDeclarationList ReturnsClause
  ( ResultCast )? RoutineCharacteristics ( RoutineCharacteristics )* RoutineBody

SQLInvokedAggregate ::=
  <AGGREGATE> SchemaQualifiedRoutineName SQLParameterDeclarationList ReturnsClause
  ( ResultCast )? RoutineCharacteristics ( RoutineCharacteristics )* RoutineBody

SQLInvokedProcedure ::=
  <PROCEDURE> SchemaQualifiedRoutineName SQLParameterDeclarationList ( ReturnsClause )?
  RoutineCharacteristics ( RoutineCharacteristics )* RoutineBody

SchemaQualifiedRoutineName ::= QualifiedIdentifier ReturnsClause ::= <RETURNS> DataType

ResultCast ::= <CAST> <FROM> DataType

RoutineCharacteristics ::=
    LanguageClause
  | ParameterStyleClause
  | NullCallClause
  | ReturnedResultSets
  | DeterministicCharacteristic
  | SQLDataAccessIndication

LanguageClause ::= <LANGUAGE> ( <JAVA> | <SQL> | <IDENTIFIER> )

arameterStyleClause ::= <PARAMETER> <STYLE> ( <JAVA> | <SQL> | <GENERAL> )

NullCallClause ::= <RETURNS> <NULL> <ON> <NULL> <INPUT> | <CALLED> <ON> <NULL> <INPUT>

ReturnedResultSets ::= <DYNAMIC> <RESULT> <SETS> <INTEGER_LITERAL>

DeterministicCharacteristic ::= <DETERMINISTIC> | <NOT> <DETERMINISTIC>

SQLDataAccessIndication ::=
  <NO> <SQL> | <CONTAINS> <SQL> | <READS> <SQL> <DATA> | <MODIFIES> <SQL> <DATA>

RoutineBody ::= ( SQLRoutineBody | ExternalBodyReference )

SQLRoutineBody ::= <RETURN> ( expression | CursorOrCallSpecification )

ExternalBodyReference ::= <EXTERNAL> <NAME> <STRING_LITERAL>

SQLParameterDeclarationList ::=
  <LPAREN> (
    [ SQLParameterDeclaration ( <COMMA> SQLParameterDeclaration )* ]
    [ … ]
  )? <RPAREN>

SQLParameterDeclaration ::= ( ParameterMode )? Identifier DataType ( <AS> <LOCATOR> )?
( <RESULT> )?

ParameterMode ::= ( <IN> | <OUT> | <INOUT> )
```

```
Identifier ::= <IDENTIFIER>

DataType ::= ( ArrayType | MultisetType | SimpleType )

SimpleType ::= (
    CharacterStringType
  | BinaryStringType
  | NumericType
  | IntervalType
  | DateTimeType
  | BooleanType
  | RowType
  | StructType
  | BlobType
  | ObjectType
  | XmlType
  | DataLinkType
  | PeriodType
  | NullType
  | AnyType
)

MultisetType ::= SimpleType <MULTISET>

RowType ::= (
    <ROW> <LPAREN> Field ( <COMMA> Field )* <RPAREN>
  | <TABLE> ( <LPAREN> Field ( <COMMA> Field )* <RPAREN> )?
)

StructType ::= <STRUCT> <LES> Field ( <COMMA> Field )* <GRT>

Field ::= Identifier DataType

CharacterStringType ::=
  ( <CHARACTER> | <CHAR> ) <VARYING> ( <LPAREN> IntegerValue <RPAREN> )?
  | ( <CHARACTER> | <CHAR> ) ( <LPAREN> IntegerValue <RPAREN> )?
  | <VARCHAR> ( <LPAREN> IntegerValue <RPAREN> )?
  | ( <NCHAR> | ( <NATIONAL> ( <CHARACTER> | <CHAR> ) ) ) <VARYING>
  ( <LPAREN> IntegerValue <RPAREN> )?
  | ( <NCHAR> | ( <NATIONAL> ( <CHARACTER> | <CHAR> ) ) ) ( <LPAREN> IntegerValue <RPAREN> )?
  | <NVARCHAR> ( <LPAREN> IntegerValue <RPAREN> )? | <STRING>

BinaryStringType ::=
    <BINARY> ( <LPAREN> IntegerValue <RPAREN> )?
  | <BINARY> <VARYING> <LPAREN> IntegerValue <RPAREN>
  | <VARBINARY> <LPAREN> IntegerValue <RPAREN>

NumericType ::=
    ( <DEC> | <DECIMAL> | <NUMERIC> )
      ( <LPAREN> IntegerValue ( <COMMA> IntegerValue )? <RPAREN> )?
  | <SMALLINT>
  | ( <INTEGER> | <INT> ) ( <LPAREN> IntegerValue <RPAREN> )?
  | <BIGINT>
  | ( <FLOAT> | <REAL> ) ( <LPAREN> IntegerValue <RPAREN> )?
  | <DOUBLE> ( <PRECISION> )?
  | <NUMBER>

IntervalType ::=
  <INTERVAL> ( ( DatetimeField ( <LPAREN> IntegerValue <RPAREN> )?
  ( <TO> ( DatetimeField | <SECOND> ( <LPAREN> IntegerValue <RPAREN> )? ) )?
    | <SECOND> ( <LPAREN> IntegerValue ( <COMMA> IntegerValue )? <RPAREN> )? ) )?

DateTimeType ::=
    <DATE>
  | <TIME> ( <LPAREN> IntegerValue <RPAREN> )? <WITH> <TIME> <ZONE>
  | <TIME> ( <LPAREN> IntegerValue <RPAREN> )?
  | <TIMESTAMP> ( <LPAREN> IntegerValue <RPAREN> )? <WITH> <TIME> <ZONE>
  | <TIMESTAMP> ( <LPAREN> IntegerValue <RPAREN> )?

DatetimeField ::= <YEAR> | <MONTH> | <DAY> | <HOUR> | <MINUTE>

BooleanType ::= <BOOLEAN>

ArrayType ::= SimpleType <ARRAY_CONSTRUCTOR_START> IntegerValue ( <RBRACKET> |
<RBRACKET_TRIGRAPH> )

BlobType ::= <BLOB> | <CLOB>

ObjectType ::= <JAVA_OBJECT> XmlType ::= <XML>

DataLinkType ::= <DATALINK> PeriodType ::= <PERIOD> <LPAREN> DateTimeType <RPAREN>
```

```
NullType ::= <NULL> AnyType ::= <ANYTYPE>
```

Notices

This information was developed for products and services offered worldwide.

This material may be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service. This document may describe products, services, or features that are not included in the Program or license entitlement that you have purchased.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Software Group
Attention: Licensing

3755 Riverside Dr.
Ottawa, ON
K1V 1B7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Depending upon the configurations deployed, this Software Offering may use session and persistent cookies that collect each user's

- name
- user name
- password

for purposes of

- session management
- authentication
- enhanced user usability
- single sign-on configuration
- usage tracking or functional purposes other than session management, authentication, enhanced user usability and single sign-on configuration

These cookies cannot be disabled.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at https://www.ibm.com/privacy/us/en/.

Trademarks

IBM, the IBM logo and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at " Copyright and trademark information " at www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies:

- Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Index